下载APP

26 | 从S4到Storm（二）：位运算是个好东西

2021-12-03 徐文浩

《大数据经典论文解读》 课程介绍 >

讲述：徐文浩
时长 20:36 大小 18.88M

你好，我是徐文浩。

上节课里，我们看到了随着时代的变迁，人们已经不满足于通过 MapReduce 这样批处理的方式进行数据分析了。于是，Yahoo 推出了 S4，不过 S4 并没有在历史舞台上站稳脚跟。在 S4 的论文发表的同一年，我们今天的主角，也就是 Storm 走上了历史舞台。在接下来的几年里，Storm 一度成为业界进行实时数据处理的标准解决方案。

令人惊叹的是，Storm 并不是来自哪一个业界的大公司。而是来自一个只有 3 个人的创业公司 BackType，它的主要作者南森·马茨（Nathan Marz）那个时候也才仅仅 21 岁。没过多久之后，BackType 就被当时如日中天的 Twitter 收购了。可以说，Storm 证明了即使是到了今天，天才工程师仍然能够凭借一己之力，对整个行业产生重要的影响。

不过，作为一个开源项目，Storm 一开始并没有以论文的形式发表。直到 2014 年，Twitter 才发表了《Storm @Twitter》这样一篇更加偏重于 Storm 如何在 Twitter 内部使用的论文。而且，这篇论文中的作者中，也没有 Storm 最初的作者南森·马茨。虽然论文里没有南森的名字，但是 Storm 最大的功劳也仍然来自于他这个最初的作者。

那么，今天我们也会不局限于论文本身的内容，而是会着重讲解 Storm 的基本原理和系统架构。在后面的课程里，我们还会进一步讲解南森提出的组合 Storm 和 Hadoop 的 Lambda 数据处理架构。

在学完这节课之后，我希望你能够理解：

Storm 的基本架构是怎么样的，对比上节课我们讲解的 S4，它解决了哪些问题，以及弥补了什么不足之处。

Storm 是如何对消息进行容错处理的，为什么 Storm 可以追踪消息的整个生命周期，但又不会过度影响整个集群的性能。

典型的 Master+Worker 架构

在上节课里，我们看到 S4 把数据的流式计算，抽象成了一个叫做 **PE** 的单元。所有的数据、处理逻辑都是基于 PE 的，整个系统也是一个没有 Master，完全对称的架构。虽然 Storm 在流式计算上，也采用了类似的有向无环图（DAG）的逻辑模型，但是在整个系统架构上，Storm 却要传统很多。

和 S4 不同，**Storm 是一个典型的 Master+Worker 的分布式系统架构**，并且将传送的消息和对应消息的处理逻辑做了分离。那么，接下来，我们就一起先来看一下 Storm 的整体架构。

基于 Topology 的逻辑模型

和 S4 类似，Storm 系统的抽象模型，也是一个有向无环图。在 Storm 里，这样一个有向无环图，叫做 **Topology**，也就是拓扑图。整个图里有这样几个元素：

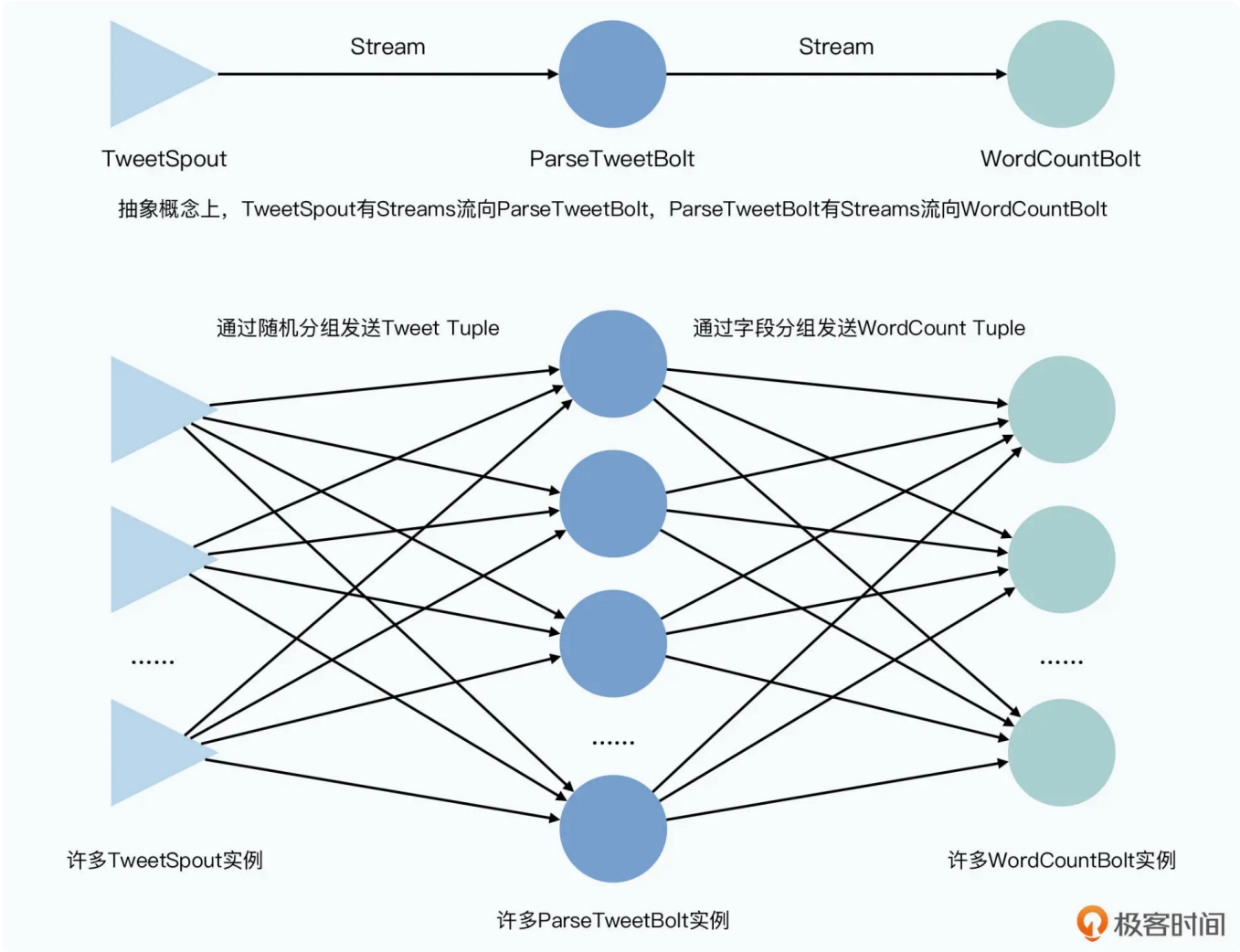
首先是 **Spouts**，也就是数据源。Storm 并没有像 S4 一样，把一切东西都定义成 PE。Spout 负责从外部去读取数据或者接收数据。就和它的名字一样，Spout 好像一个出水

管，一旦打开，就会源源不断地有外部的数据灌进来。在 S4 里，对应的就是无键 PE（Keyless PE）。

其次是 **Tuple**，也就是元组。它也是我们在 Topology 中，传输的所有的最小粒度的数据单元。一个 Tuple 是一个带命名的值的列表，你可以看成是一个个 KV 对，不过这个 Key 只是在定义 Tuple 的时候出现。但是在数据传输的时候，我们只需要传输对应的值。这个方式，其实有点像我们之前讲解过的 Thrift，字段的名称是定义在外部的，实际传输的时候只需要序号、类型和值。在 S4 里，Tuple 对应的就是事件（Event）。

然后是 **Streams**，也就是数据流，一个流包含了无限多个 Tuple 的序列，这些 Tuple 会被系统分布式地并行去处理。

最后就是 **Bolts**，也就是我们进行计算逻辑处理的地方。Bolt 可以处理任意数量的输入流，然后产生任意数量的输出流。对应地，我们要把计算结果写入到外部数据库，也是通过 Bolts 来进行处理。乍一看，Bolts 似乎很像 S4 里的 PE，用于处理对应的分布式计算逻辑，不过实际上，Bolts 和 PE 完全不一样。



论文中的图1和图3的重绘，和S4的PE不同，Storm的Bolt只是逻辑处理单元，而不是每一个Key就有一个Bolt对象

对于数据的分发，是通过对于Tuple的Grouping来进行的，所以系统的并行度可以设置，而不是和PE对象数量相关

Storm 的抽象模型里，和 S4 的最大不同就在 **Bolts** 上。S4 的 PE，不仅是一个功能逻辑的单元，也是一个 KV 对的数据。同样类型的事件下，所有相同的 Key 的数据，都会聚合到同一个 PE 下。这就使得整个系统里有大量的 PE 对象，也导致 S4 的整个系统有几个显著的设计问题。

首先就是**内存占用和 GC 开销，大量的 PE 会占用大量的内存**。

但是，这个内存占用又是应用开发人员完全控制不了的。因为系统里有多少 PE，以及当前计算节点的内存占用是否过大，是由 S4 框架控制的。而对于应用开发人员来说，他操作的只是单个 PE 对象，显然单个 PE 对象本身不会占用太多内存。在内存不足的时候，我们原本可以在应用层面有更灵活的操作，比如更频繁地把数据输出到外部的 KV 数据库里，释放掉内存占用。但是在 S4 的框架下，我们做不到这一点。

其次就是我们的**业务逻辑代码里，混入了控制分布式数据分发的逻辑**。

我们在上节课的代码里面看到了，为了让 Top K 的排序能够分布式地并行执行，对应的示例代码中，特地将输出的 Key 变成了（SortID，N）这样的组合。也就是靠 PE 里的逻辑处理代码，来设置整个拓扑图的并行度。这使得我们分布式的分发逻辑，和数据处理逻辑混合在了一起，整个系统被耦合起来了。

而且，如果我们的数据量增加了，想要提升并行度，我们不能简单地修改参数，而是需要修改代码，重新编译部署。并且，历史上已经处理了（SortID, N）组合 PE，在重新部署之后，处理的数据可能是完全不同的，因为我们的 N 的最大值已经变了。

而 Storm 的设计并不相同，**Storm 里的 Bolt 更像是 MapReduce 里的 Map 或者 Reduce 函数**。我们可以在 Topology 里面，去设置不同 Bolt 的并行度，以及设置数据流是如何分组的。但是，每个 Bolt 输出的 Tuple 本身，却不需要通过生成一个类似于（SortID, N）这样一个特殊的 Key，来定义下一层的 Bolt 的并行度。在 Storm 里面，对应的数据流可以进行这样几种分组（Grouping）：

随机分组（Shuffle Grouping），也就是每一个 Bolt 输出的结果，在分发到下游的 Bolt 的时候，是随机分发给不同的 Bolt 的，每个 Bolt 都会收到数量接近的 Tuple。

字段分组（Fields Grouping），可以选定 Tuple 中的某一个字段，按照字段的值进行分组。比如我们如果还是要进行单词出现频率的 Top K 排序，我们可能有一个 Tuple，第一个字段叫做 word，存放具体的单词，第二个字段叫做 count，存放它出现的频率。那么，我们可以通过 word 这个字段进行字段分组，这样，相同单词的 Tuple 就会分发到相同的 Bolt 里去了。

全部分组（All Grouping），这个类似于数据广播，也就是 Bolt 输出的 Tuple 需要向下游的每一个 Bolt 都发送一份。

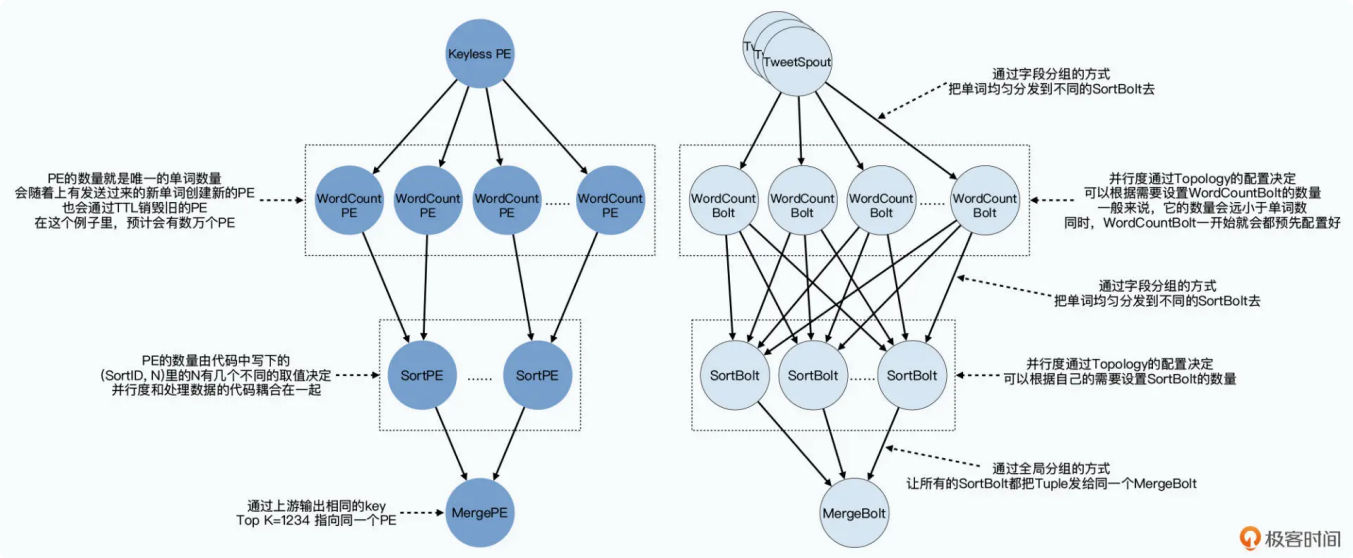
全局分组（Global Grouping），这个类似于 S4 里你看到的终点的 MergePE，所有上游的 Bolt 都会把 Tuple 发送到唯一一个下游 Bolt 中。这样在下游，就可以有全局信息来做统计判断。

无分组（None Grouping），这个是说开发人员不关心这个数据怎么分组。在实际实现里面，它和随机分组是一样的。

指向分组（Direct Grouping），这个是指，上游的 Bolt 可以指定下游由哪一个 Bolt 来接收对应的 Tuple。

本地或随机分组（Local or Shuffle Grouping），也就是当下游的 Bolt 如果有一个或者多个“任务（Tasks）”，和上游的 Bolt 在同一个 worker 进程里，那么 Tuple 只会分发到这些进程里的任务里。如果没有的话，那就还是按照随机分组的方式发送 Tuple。这个主要是为了性能考虑，如果可以在同一个台机器的同一个进程内通信，会大大节省整个集群的网络开销。

如果你对照着这里的示意图，可以看到在 Storm 下进行 Top K 的单词排序，两边的数据流向是不一样的，S4 里的一个 WordCountPE 的输出，只会给一个 SortPE；而 Storm 里的 WordCountBolt 的输出，会发送给多个不同的 SortCountBolt，因为同一个 WordCountBolt 下，会包含很多个不同的单词。



看似S4和Storm的拓扑结构类似，但是实际上PE和Bolt是完全不同的概念
Bolt并不会随着新单词的出现创建，也不会随着TTL而销毁

Master+Worker 的系统架构

我们说 Storm 的 Bolt 很像 MapReduce 的 Map 和 Reduce 函数，其实 Storm 本身的架构也和 MapReduce 非常相似。和我们上节课看过的无中心的 S4 不同，Storm 选择了一个典型的 Master+Worker 的架构设计。整个 Storm 集群里，也是由 **Nimbus+Supervisor+Worker** 这样三种类型的进程组成的。

首先是 **Nimbus 进程**，其实也就是 Storm 集群的 Master 节点。它的作用，类似于 Hadoop 里的 JobTracker，或者说 MapReduce 里的 Scheduler+Master，也就是负责资源的分配和任务的调度。

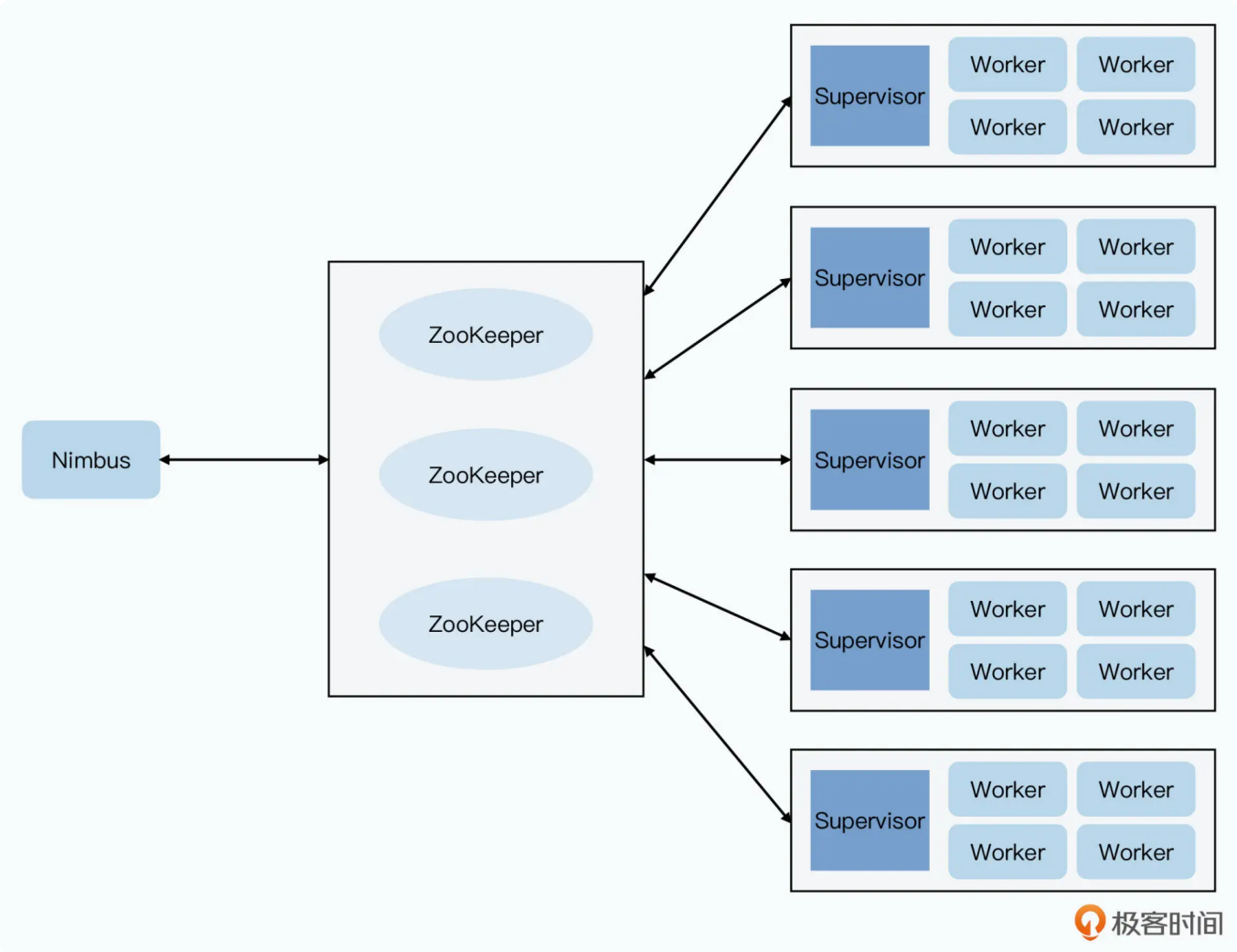
开发人员会直接提交一个 Topology 给 Master。这个 Topology，之前只是一个抽象的有向无环图。而在实际应用里，它就好像一个 MapReduce 的任务一样，是一个编译好的程序和对应的配置。只不过，MapReduce 的任务执行完了就结束了。而作为流式计算，Topology 这个任务如果我们不去终止它，它就会永不停歇地运行下去。

然后是 **Supervisor 进程**，这个类似于 Hadoop 里的 TaskTracker，也就是 MapReduce 里的 Worker。Supervisor 在每一个服务器上都会有一个，它本身不负责执行任务，但是会负责接收 Nimbus 分配的任务，然后管理本地的 Worker 进程，让 Worker 进程来实际执行任务。

最后是 **Worker 进程**，一台服务器上会有多个 Worker 进程。Storm 是使用 Clojure 写的，跑在 JVM 上，所以每一个 Worker 进程就是一个独立的 JVM，Worker 里面还会通

过 JVM 的 Executor 来维护一个线程池。然后实际的线程池里，会有很多个 Spout/Bolt 的任务。因为 Java 的 Executor 的实现里会复用线程，所以 Spout 和 Bolt 实际上会使用同一个线程。这个，也会大大减少整个系统的开销。

而把整个系统拆分成 Nimbus、Supervisor 和 Worker 三种进程，就使得 Storm 的容错能力也大大增强了。



论文中的图2，Storm系统的宏观架构图，我稍稍做了一下修改，方便你理解每个物理服务器里会有多个Worker进程

Nimbus 和 Supervisor 之间，并不是直接通信的。因为如果这样的话，显然 Nimbus 会成为一个故障的“单点”。所以 Nimbus 是把对应的任务分配写到 Zookeeper 里，也就是一个类似于 Chubby 这样的分布式锁系统。所以我们的任务分配是**持久化**的，而且会由 Paxos 协议来保障容错能力。而 Supervisor 也是从 Zookeeper 里面，去读取对应的任务分配。

Nimbus 和 Supervisor 的职责都非常简单，Nimbus 只需要进行 Topology 的解析和任务调度，而 Supervisor 只需要接收任务，并且监控 Worker 进程是否存活。**它们本身不处**

理数据，而且也不在内存里面保存数据。即使挂掉了，也只需要简单重启一下进程就好了。

这种类似的设计思路，我们在 Megastore 里的协同服务器（Coordinator）里已经看到过一次了，通过把一些职责单独拆分出来，让特定的节点足够简单。即使这些节点可能成为单点，但是它们的稳定性，也会远高于要处理复杂逻辑的 Worker 进程或者 Bigtable。

作用	Storm	MapReduce	Hadoop
主节点	Nimbus	调度器+Master	JobTracker
工作节点	Supervisor	TaskTracker	Worker
处理数据进程	Worker	N/A	Child
计算任务	Spout/Bolt	Map/Reduce	Mapper/Reducer
提交的程序	Topology	Job	Job

极客时间

其实Storm对于系统的拆分和MapReduce非常相似

这样来看，Storm 的整体设计思路和 MapReduce 很像，各个节点的角色都能在 MapReduce 的各种节点里找到对应的影子。其实，**各类分布式系统的设计思路都是类似的**，特别是这样 Master+Worker 组合的模型，那就是 Master 负责调度，Worker 负责实际处理问题。而为了解决高可用性，往往我们会引入分布式锁，确保任务分配的数据不依赖 Master。

另外，为了让整个系统更稳定，我们也会拆分调度任务的进程，和直接执行任务的进程，让每个进程都只有单一的职责。希望在学习和解读完这么多篇论文之后，你在未来的系统设计里。也能遵循这样的理念。

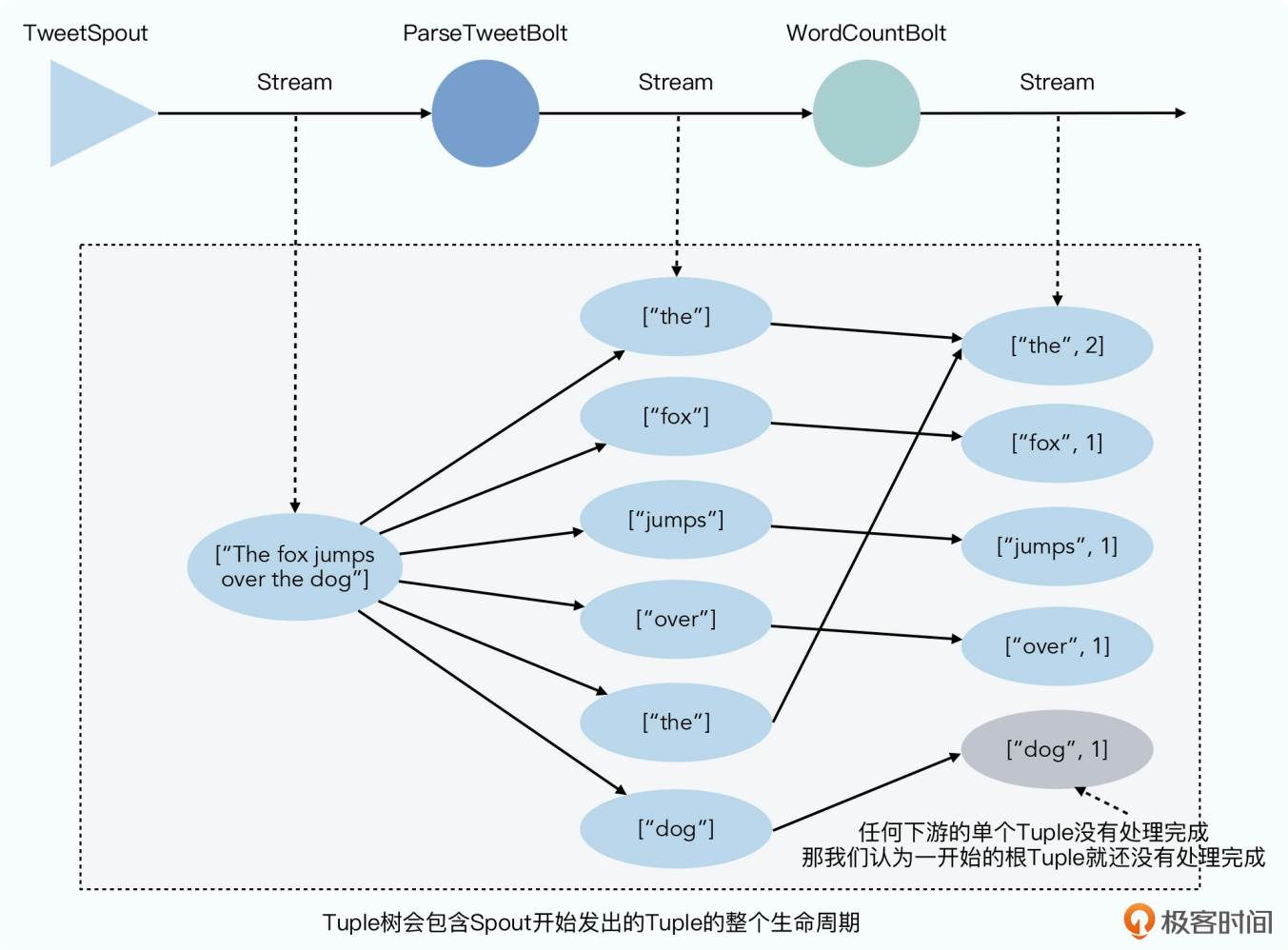
Storm 的容错机制

在 S4 的论文里，Yahoo 并没有详细说明它的上下游是怎么通信的。不过在 Storm 里，我们倒是可以通过源码很清楚地知道，它是通过 **ZeroMQ** 这个消息队列，完成两个不同的 Worker 之间的通信的。

相比于通过一个 RPC，消息队列有一个很大的优点，那就是**高性能**。上游节点不需要等待下游节点返回接收成功，就能发送下一条信息。不过，这也带来了一个问题，就是如果在消息发送之后，下游是否成功接收并处理了这条消息，上游是不知道的。可能因为网络超时、也可能因为下游节点的软硬件故障，**在分布式系统里，“错误”是在所难免的**。

而且，在流式数据处理里，我们可能不只有一层链路。就拿论文里的统计 Tweet 里的单词数量为例，我们先要从一个 TweetSpout 里，读取数据流里的 Tweet，随机发送给到一个 ParseTweetBolt，这个 Bolt 会解析 Tweet 成一个个单词，再发送给下游的多个 WordCountBolt。

要注意，这里下游不只有一个 WordCountBolt，而是不同的单词会发送给不同的 WordCountBolt。而且，任何一个 WordCountBolt 没有被成功处理，都意味着我们面临“错误”。



一个从Spout发出的Tuple，后续所有衍生出来的Tuple都属于这个根Tuple树的一部分

Storm 选择的解决方案，是把从 Spout 发起的第一个 Tuple 作为一棵树的根。下游所有衍生出来发送的 Tuple，都是这棵树的一部分。任何一个 Tuple 处理失败或者超时了，那

么就从 Spout 重新发送消息。

而要做到这一点，Storm 需要在系统里引入一个特殊的 Bolt，叫做 AckerBolt。Spout 发送出去的消息，同时会通知给到 AckerBolt。而 Bolt 一旦处理完**根 Tuple** 相关的消息，也会通知给到 Acker。

Bolt 会告诉 AckerBolt 两个信息，一个是我已经处理完了某一个 Tuple，另一个是这个 Tuple 衍生往下游的哪些 Tuple 我也已经发送出去了。这样，Acker 就有了一开始 Spout 发出的 Tuple 的整棵树的完整信息。等到最后一层的 Bolt 处理完对应的 Tuple，然后发送了对应的通知给到 AckerBolt，并且告诉它后面没有新的 Tuple 了，那么 AckerBolt 就知道，整棵 Tuple 树已经处理完成了。

相信看到这里，你会和我有一个同样的疑问，那就是如果这样，AckerBolt 的开销岂不是会非常大？

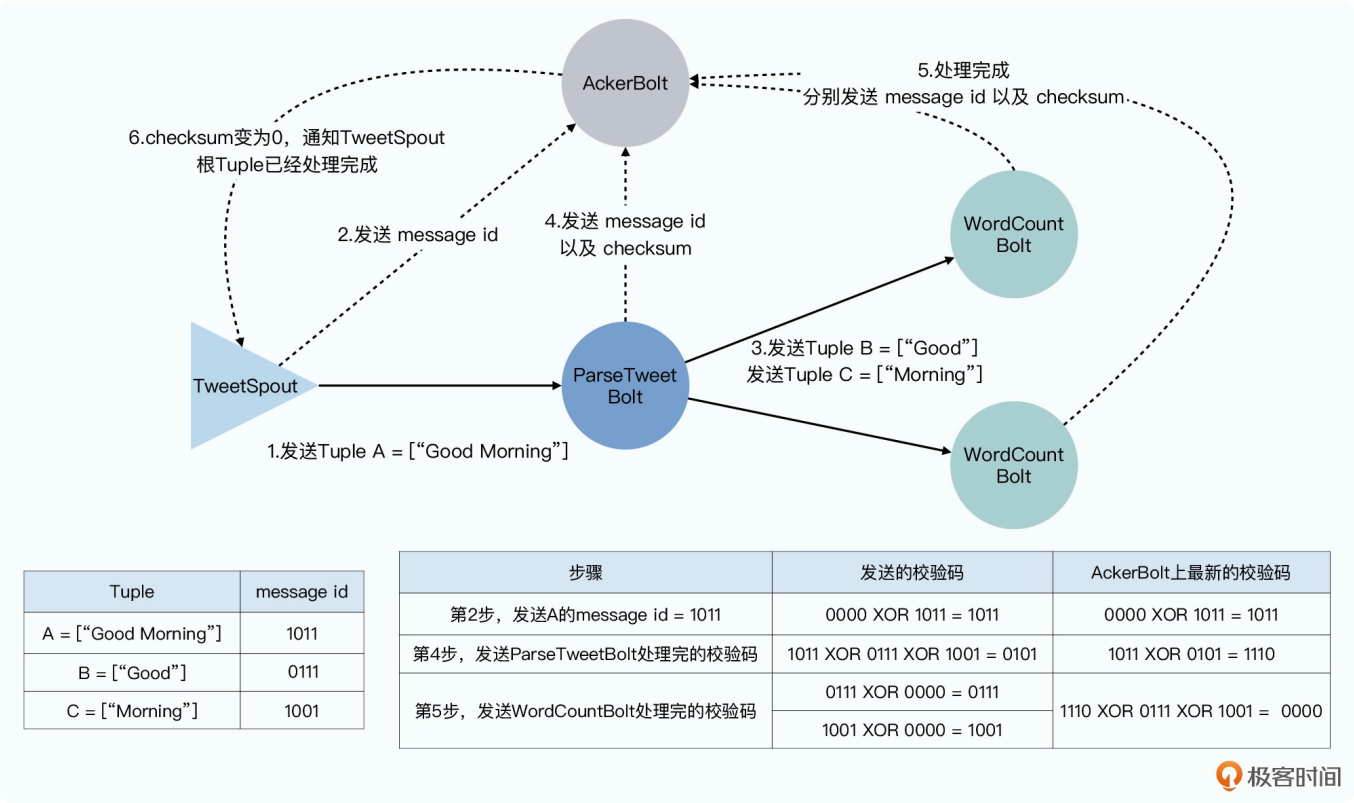
因为看起来，我们要在 AckerBolt 里面，存上整棵 Tuple 树。更准确地说，AckerBolt 不是要存一棵 Tuple 树，而是要把所有还在处理中的 Tuple 都存下来。这就相当于一个 AckerBolt，需要存下所有 Spout 和 Bolt 在整个处理过程中的内存占用，这样我们的集群哪里抗得住？

别着急，Storm 采用了一个很巧妙的办法，那就是**利用位运算里的异或（XOR）**。Storm 给每一个发送出去的 Tuple 都会分配一个 64 位的 message id。当消息从 Spout 被发送出去的时候，Storm 会给 AckerBolt 发送这个 message-id，告诉它，你要开始追踪这个 Tuple 树了。Acker 里呢，则会维护一个 message-id 到校验码（checksum）的映射关系。这个校验码，一开始就是拿 0 和 message-id 去异或（XOR）一下。

而下游的每一个 Bolt，会处理完这个 Tuple 相关的消息，并且向外发送新的 Tuple。每个新发送的 Tuple 里，都需要带上根 Tuple 的 message-id。在新 Tuple 发送出去之后，Bolt 会通知 AckerBolt，通知的内容也很简单，也是一个根 message-id 到校验码的映射关系。

这里的校验码，就是把当前对外发送的所有消息的 message-id，和已经处理完的消息的 message-id 做一下异或。然后 AckerBolt 收到这个消息，会把收到的校验码，和本地的校验码也做一下异或，更新成最新的校验码。

因为异或操作，就是当两个数字完全相同的时候，会变成 0，也就是 A XOR A = 0。而在其他情况下，最后的结果一定不会每一位都是 0。我们发送一次消息，并且 acking 一次消息，相当于在这个校验码上执行了一次 A XOR A。所以，只要有 Tuple 还没有被 acking，我们的校验码就不会是 0，但是一旦所有的 Tuple 树上的 Tuple 都被 acking 了，那么这个校验码必然就是 0。



Tuple	message id
A = [“Good Morning”]	1011
B = [“Good”]	0111
C = [“Morning”]	1001

步骤	发送的校验码	AckerBolt上最新的校验码
第2步，发送A的message id = 1011	0000 XOR 1011 = 1011	0000 XOR 1011 = 1011
第4步，发送ParseTweetBolt处理完的校验码	1011 XOR 0111 XOR 1001 = 0101	1011 XOR 0101 = 1110
第5步，发送WordCountBolt处理完的校验码	0111 XOR 0000 = 0111	1110 XOR 0111 XOR 1001 = 0000
	1001 XOR 0000 = 1001	

极客时间

你可以随机生成一些message-id，按照这个XOR的算法试一下，看看是否所有消息处理完后，对应的校验码就是0

通过 message-id 加上校验码，Storm 只需要 16 bytes 就能在 AckerBolt 里维护一个 Tuple 树是否已经都处理完了。这样，即使你每秒需要处理 10 万条消息，AckerBolt 里需要维护 30 秒的 Tuple，也只需要 48MB 的内存空间，这即使对于 10 年前的服务器来说也是绰绰有余的了。

而且，所有的 Bolt 通知 Acker 最新的执行情况，也只需要发送 16 bytes 的 messsage-id 和校验码，既不需要发送 Tuple 的原始内容，也不需要为向下游发送的每一个 Tuple 都单独发送一条消息，占用的性能消耗也不会太大。

不过，需要注意的是，**这个机制只能保障，Spout 发出来的 Tuple 至少被处理一次，也就是 At Least Once，但是它避免不了 Tuple 可能被重复处理。**

比如，拿我们的 Top K 排序的 Topology 来说，任何一个单词在某个 Bolt 里没有被正确处理，就需要重新处理整个句子。这也意味着，其他单词会被重复统计。所以，这个通过 AckerBolt 进行容错重发的机制，并不适合所有的应用场景。你需要根据自己的实际业务需求，来决定要不要启用这个机制。

有了 At Least Once，那你自然会想到还有 At Most Once，也就是一条消息最多发一次。这个要求其实在 Storm 里很容易做到，我们只要关掉这里的 acking 机制就好了。其实也就是上游只需要把消息发出去，下游有没有收到、有没有处理成功，上游就不管了。

然而，无论是 At Least Once 还是 At Most Once，都不是我们最理想的进行流式数据处理的方式。我们真正希望的，是每个消息“恰好”被处理一次，也就是“**Exactly Once**”。所以，Storm 还不是流式数据处理最终极的解决方案。在 Storm 之后，整个流式数据处理系统还会不断进化。

小结

好了，到这里，我们对于 Storm 的剖析就已经告一段落了。今天，我们一起梳理了 Storm 里的系统概念、整体的框架设计，以及它是如何巧妙地通过异或运算，来追踪整个 Tuple 的生命周期的。

在系统设计层面，Storm 其实和我们之前看过的 MapReduce/Bigtable 之类的系统，有很多相似之处。Storm 通过 Nimbus 这个主节点进行任务调度，通过 Zookeeper 存储所有的元数据，通过把工作节点拆分成 Supervisor 和 Worker，来提升系统稳定性。可以说，太阳底下没有什么新鲜事儿。

而且，这样的系统架构，也的确比无中心的 S4 更加合理。不同于 S4 把数据和处理逻辑绑定成 PE，并且连整个系统的并行度，也要依靠逻辑代码中选取的 Key 来决定。Storm 完成遵循了“**单一职责**”这个模式，并且让分布式的并行度、数据的有向无环图，彻底和逻辑代码拆分开来了。

进一步地，Storm 还开始针对业务层面的容错做出了更多考虑。相比于 S4 只是简单地在高压力下进行降级，Storm 开始深入考虑一个分布式流式系统，应该怎样对于没能成功处理的消息进行错误恢复。通过巧妙地利用**异或运算**，Storm 可以用很小的内存和网络开销，来追踪整个 Tuple 处理的生命周期，并通过重发消息实现“至少一次”的消息处理保障。

那么，在接下来的课程中，我们还会再深入其他的流式数据处理系统，一起去看一看流式数据处理会如何逐步走向完善。

推荐阅读

随着 Storm 登上历史舞台，如何把 Storm 这样的流式计算和 MapReduce 这样的批处理系统统一起来，也成为了一个热点话题。而 Storm 的作者南森本人，也提出了一个叫做 Lambda 架构的解决方案，Twitter 后续也围绕这个架构，开发了 Summingbird 这个系统。

对于 Lambda 架构，南森·马茨本人在他写的《大数据系统构建》这本书里，做了非常详尽的讲解，如果有时间的话，我推荐你可以去快速翻阅一下这本书。

虽然在 2021 年的今天，你可能已经用上了 Kafka、Flink，甚至觉得 Lambda 架构已经有些过时了。但是**在整个系统开发中，不只是考虑系统硬件的故障，而是考虑到开发人员本身也会出错，也会写 Bug，这是 Lambda 架构中最重要的核心思想**。从这一点来说，Lambda 架构在今天仍然有着它独特的价值。

另外，我们也会在后续讲解 Kafka 的过程中，来一起看看 Kafka 作者杰伊·克雷普斯（Jay Kreps）提出的 Kappa 架构，和南森·马茨提出的 Lambda 架构之间的对比。

思考题

Storm 里，我们需要在 Topology 提交的时候，就设定好各个 Bolt/Spout 的数量，也就是整个系统的并行度。那么，如果遇到突然增加的流量，我们是否会遇到和 S4 一样的问题，不得不随机丢掉一些消息呢？我们可以做哪些工作，确保整个 Storm 系统在出现流量陡增的情况下，仍然可以通过添加硬件进行水平扩展呢？

欢迎在留言区分享你的答案和思考，也欢迎你把今天的内容分享给更多的朋友。感谢收听，咱们下节课再见。

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 0 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 25 | 从S4到Storm（一）：当分布式遇上实时计算

训练营推荐

精选留言（1）

写留言

在路上
2021-12-03

徐老师好，我认为可以通过修改Topology来提高Bolt的并行度，新流入的数据根据新版的Topology分发数据。Bolt节点分成两种，一种是无状态的，一种是有状态的。比如ParseTweetBolt是无状态的，很容易扩展，WordCountBolt是有状态的，需要把状态迁移到新节点上，可以采用一致性hash扩容、翻倍扩容等方式，减少需要迁移的数据。

展开